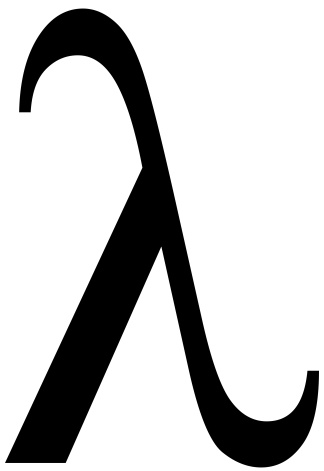
Lambda Expressions in Java

Erhan Bagdemir

bagdemir.com - Follow on @ebagdemir

February 11, 2015

Today's Agenda



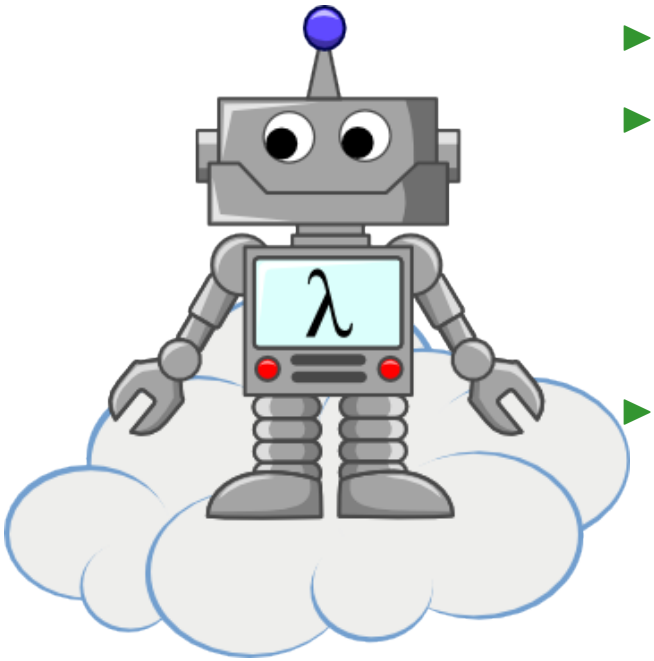
- Definition of Lambda.
- Lambda Expressions in Java.
- Functional Interfaces
- Method References :: Operator
- forEach()
- Streams

Lambda Expressions: Definition of Lambda

- **A formal system for expressing computational behaviour**.
  - Invented by Alonzo Church in 1930.
- Lambda expressions consist of many parentheses i.e in **Y-Combinator**:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Lambda Expressions: The Idea, behind



- **Functions are first-class citizens.**
- **Lambda expressions are high order functions:**
  - They take other functions as a parameter.
  - They may return functions.
- **The functions in Lambda all referentially transparent (pure functions). They:**
  - provide a better parallelisation (no side-effects),
  - are easier to test,
  - are cacheable and provide *lazy* evaluation.

Lambda Expressions: Java 8

Some examples of function definitions:

- in Javascript : **function () { return x + 1 };**
- in LISP: **(lambda (x) x + 1)**
- in C++11: **[](int x) { return x + 1; }**
- in Scala: **x => x + 1 or just _ + 1**
- in Java 8: **(int x) -> x + 1**

Lambda Expressions: Type of Lambda Expressions

Types of lambda expressions are defined in the **java.util.function**

```
Function<Integer, String> toStr = x -> Integer.parseValue(x);
```

You can pass lambdas as a parameter:

```
final Double forNetPrice = 50.0d;
final Function<Double, Double> withVAT = x -> x * 1.19;
prepareForShipment(withVAT, forPrice);
```

Lambda Expressions: Type of Lambda Expressions

If there is no functional interface in the **java.util.function** package for your usage, create your own:

```
@FunctionalInterface
public interface WorkflowLambda {
    IAsset execute(String clientId, String userId, MetadataView
        view);
}

WorkflowLambda workflow = (clientId, userId, view) ->
    executeWith(clientId, userId, view);
```

- Call *Functional Interfaces* or *Single Abstract Method* interfaces.

Lambda Expressions: Scope of Lambda Expressions

The scope of lambda expressions, the scope of the enclosing type:

```
@Service
public class OrderManagement {

   @Inject
   private ShippingCostService scService;

   private void calculateOrder() {
     /* final */ double vatRate = vatService.getVAT(Country.DE);
     Function<Double, Double> total = x -> (x * vatRate) +
         this.scService.getCost();
   }
}
```

Local variables can be referenced by lambda expressions, as long as they are **effective final**.

Lambda Expressions: Method References

We can use methods of existing classes as lambda expressions.
Method references are syntactic shortcuts to the lambda expressions:

```
// static <T> void sort(T[] a, Comparator<? super T> c)

Collections.sort(myList, (firstInt, secondInt) -> firstInt -
    secondInt);
Collections.sort(myList, FooComparator::compare);

// if we'd an existing comparator
public class ExistingComparator {
   public static Integer compare(Integer first, Integer second) {
        return first - second;
   }
}
```

Lambda Examples: forEach()

- Collections know how to iterate through their elements.
- It's a better style, which provides functional polymorphism, in comparison to imperative external loops.

```
final List<Integer> myList = new ArrayList<>(3);
myList.add(1);
myList.add(5);
myList.add(2);

myList.forEach(System.out::println);
```

A new method in the "Collection" interface without breaking code?

Lambda Examples: Default Implementations

Interfaces may have **default** implementations:

```
public interface Dog {

    public void wagTail();

    public default void bark() {
        System.out.println("Bark!");
    }
}
```

- The interfaces may contain static methods as well.

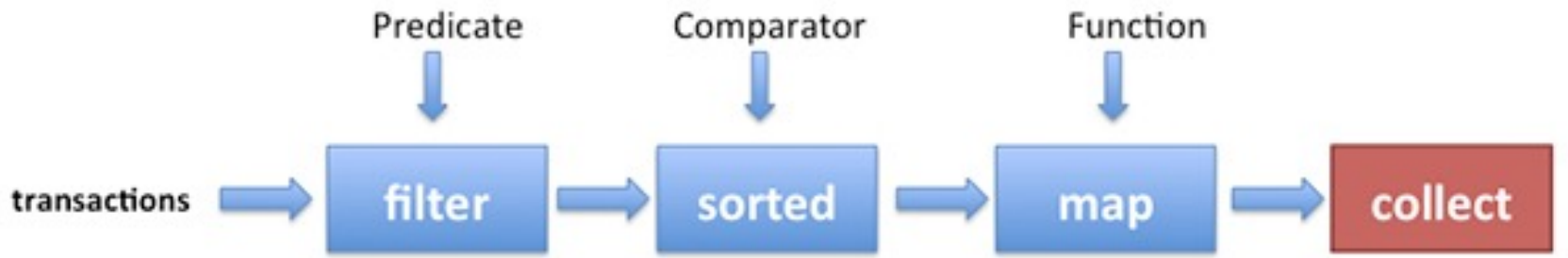
Streams

Lambda Examples: Streams

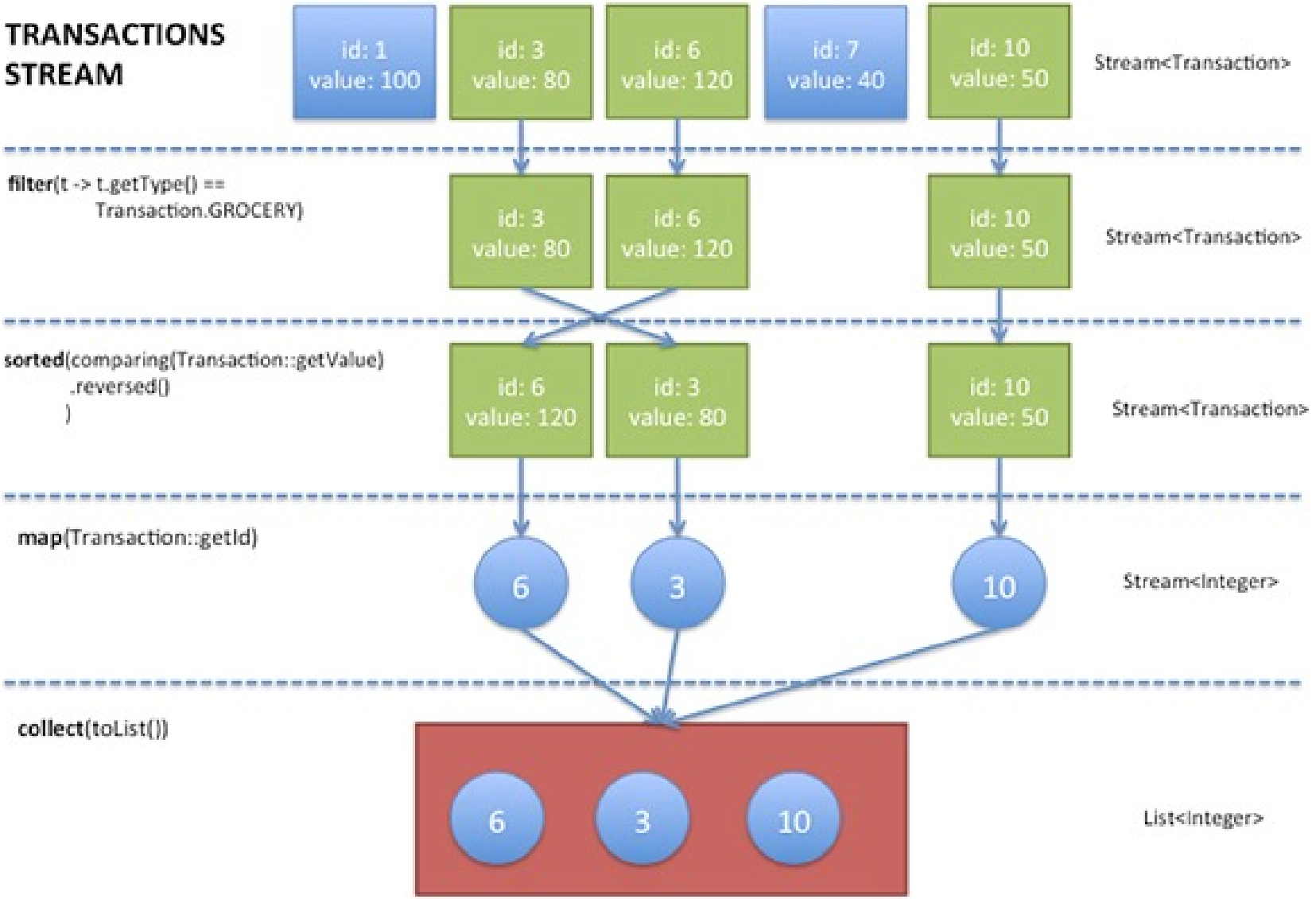
- Streams represent a sequence of elements, which support different operations that perform computations on these elements.
- There are two types of operations:
  - Intermediate Operations (filter, sort, etc.).
  - Terminal Operations (collect, forEach, reduce, etc.)
- Stream features:
  - No storage.
  - Functional in nature (No side-effects).
  - Streams are lazy (except sort).
  - Allowed to be unbounded.
  - Consumable.
- Streams are monads!

Lambda Examples: Stream Pipelines

The processing pipeline consists of a source, intermediate operations and a terminal operation.



Lambda Examples: Stream Processing



Lambda Examples: Streams

- Stream sources: Arrays, Collections, Generators and I/O Streams, e.g:

```
try (BufferedReader br = new BufferedReader(new
     InputStreamReader(is))) {
    br.lines().forEach(System.out::println);
}
catch (Exception e) {
    System.out.println(e.getMessage());
}
```

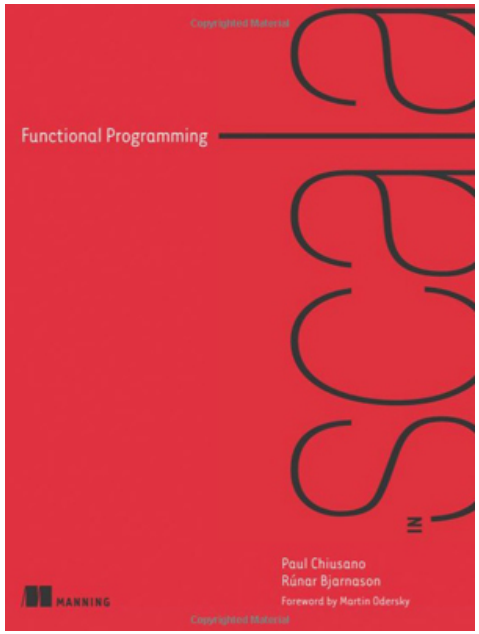
- After a terminal operation, the stream is considered to be consumed.
- Intermediate operations are lazy and always return a new stream. (see Example 6)

Lambda Examples: Code Samples

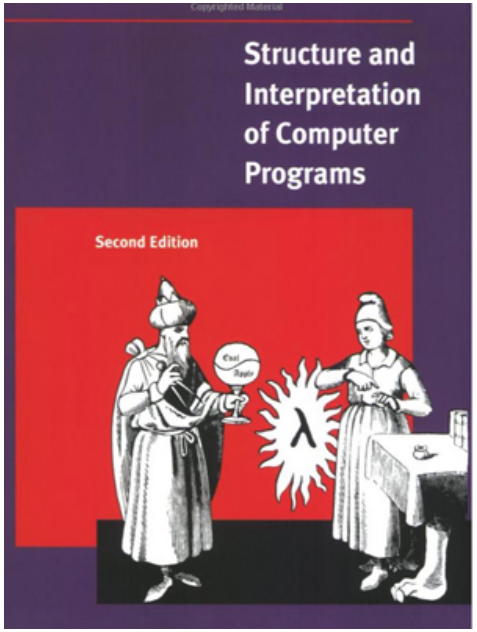
Some examples (https://github.com/bagdemir/java8-training) :

- Generators
- Comparators
- File I/O
- Regular Expressions
- Reducer
- Collectors
- Parallel Streams

Lambda Examples: Finish



Functional Programming (in Scala)



Structure and Interpretation of Computer Programs

Links

Lambda Quick Start

Oracle.com : http://bit.ly/1eR0we0

-

Streams

Oracle.com : http://bit.ly/1eh4aZo

Oracle.com : http://bit.ly/1ustKrM

-

Contact

https://github.com/bagdemir

https://twitter.com/ebagdemir

http://www.bagdemir.com